Informe Taller 1: Funciones y Procesos

Manuel Rosero - 2176007
Karol Tatiana Burbano Nasner - 2359305
Sebastian Castro Rengifo - 2359435

16 de septiembre de 2024

1. Informe de Procesos

En esta parte del informe se presentan los procesos generados por los programas recursivos, se generan algunos ejemplos para cada ejercicio, y muestre como se comporta el proceso generado por cada uno de los programas.

1.1. Máximo de una lista de Enteros

1.1.1. Función para el Máximo de una lista maxLin()

Ejemplo para maxLin(List(1,2,3,4,5))

Este es el inicio del primer ejemplo en el que le asignamos una lista de tamaño 5 a maxLin, con los enteros 1, 2, 3, 4 y 5. Como se puede observar ya se ha podido iniciar la recursión, al evaluar esta de ultimo por lo que posteriormente retornará el valor para la siguiente evaluación de maxLin.

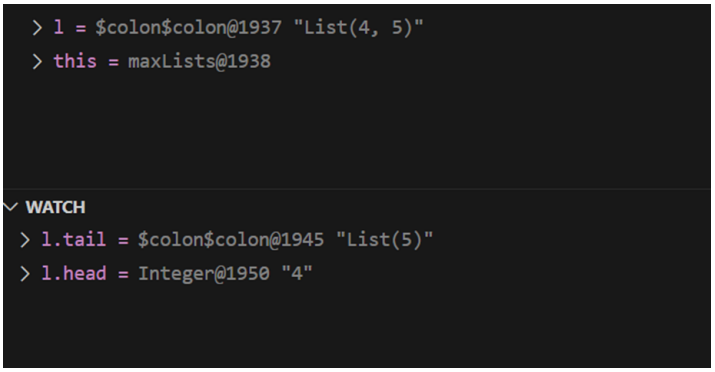


Figura 1: Debug maxLin 1

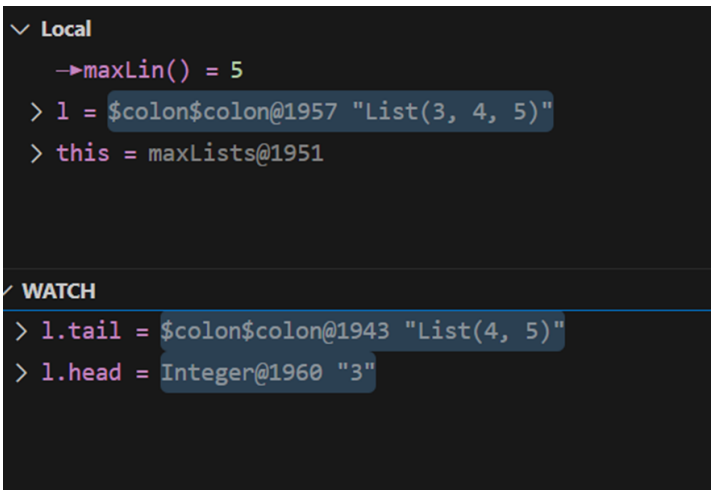


Figura 2: Debug maxLin 2

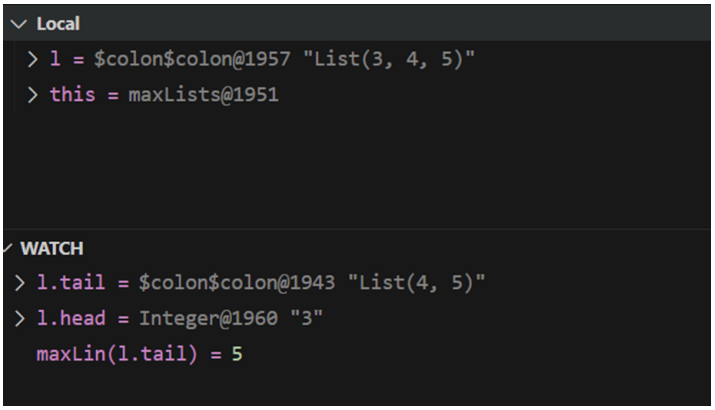


Figura 3: Debug maxLin 3

Primero retorna el valor de maxLin(4,5) como se mencionó antes para luego evaluar el método con el valor de la cabeza previa siendo 3, Por lo que posteriormente se realizara la evaluación del condicional y esto nos debe de retornar que maxLin(3,4,5) es igual a 5.

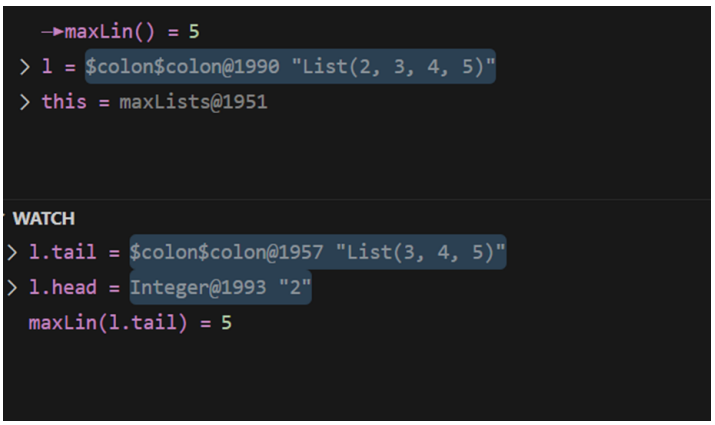


Figura 4: Debug maxLin 4

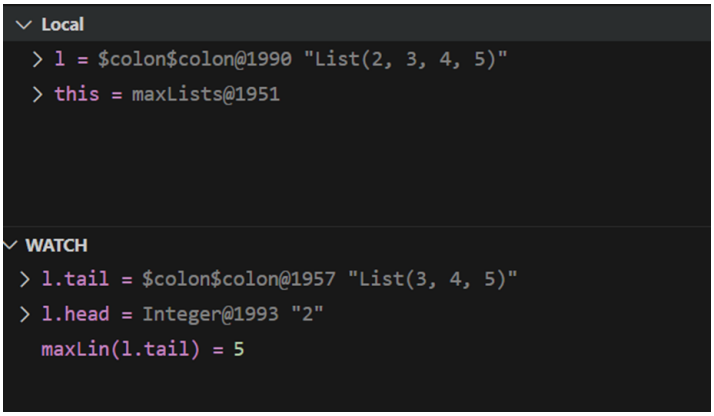


Figura 5: Debug maxLin 5

Posteriormente se asigna al valor de maxLin(l.tail) el valor anteriormente dado del método aplicado a maxLin(3,4,5) ahora comparándolo con la Head de la lista con el 2 ya que se evaluó, dando lugar a que le asigne el resultado a la siguiente evaluación del método luego de pasar el condicional

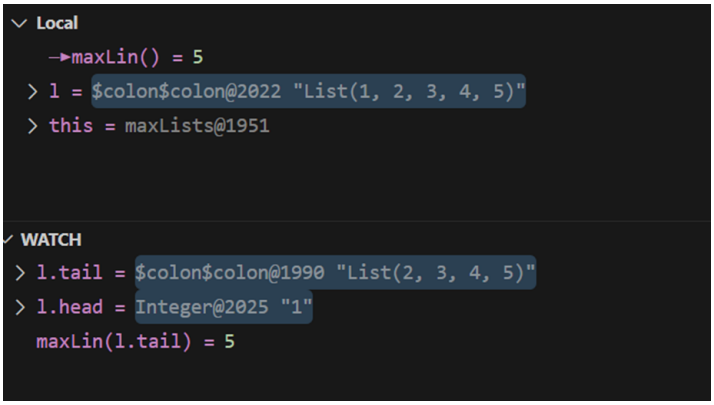


Figura 6: Debug maxLin 6

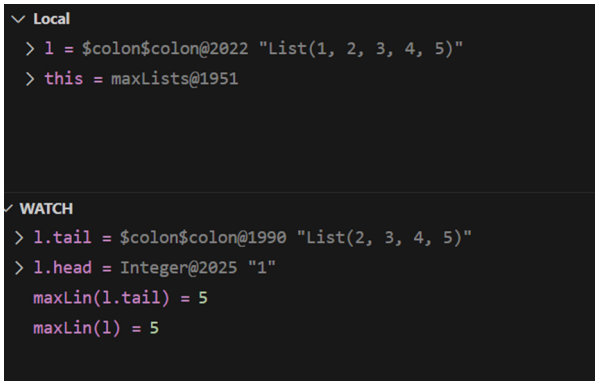


Figura 7: Debug maxLin 7

Finalmente el código se llama a si mismo al final de la recursión para poder evaluar el resto de la lista y el Int que devuelven con la Head de toda la lista, una vez que el programa ha hecho la evaluación final entonces retorna el valor Int que cumple con la condición de ser el mayor numero de la lista. En este caso siendo el 5.

1.1.2. Función para el Máximo de una lista maxIt()

Ejemplo para maxIt(List(7,2,3,9))

La función maxIt recorre recursivamente la lista (7,2,3,9), comparando cada elemento con el máximo actual, que inicialmente es 7. Al llegar a 9, se actualiza el máximo. La lista no está vacía durante este proceso, por lo que l.isEmpty es false. maxIt(l.tail) y maxIt(l) devuelven 9, el mayor número de la lista.

Figura 8: Debug maxIt 1

Como se puede ver primero se asigna a l la lista a la que se le quiere encontrar el maximo valor por lo que apenas empieza separando el head y su tail

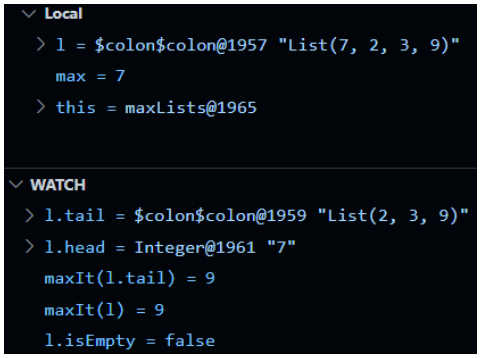


Figura 9: Debug maxIt 2

Aquí, la función ha avanzado a la siguiente iteración. La lista actual es (2, 3, 9), pero max sigue siendo 7. l.tail ahora es [3, 9] y l.head es 2. maxIt(l.tail) y maxIt(l) siguen devolviendo 9, manteniendo el máximo encontrado.

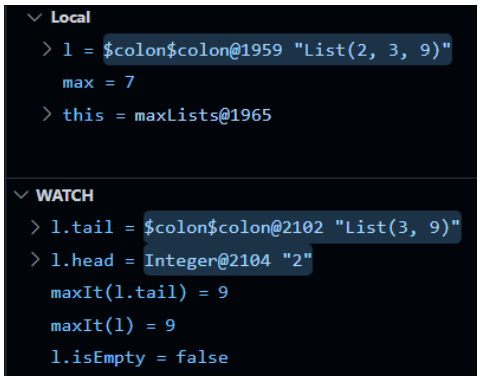


Figura 10: Debug maxIt 3

Posteriormente la función se llama de manera recursiva lo con el resto de la lista, ahora afectando a (2,3,9) al evaluarlo, el max sigue siendo 7 así que por el condicional retorna l.head

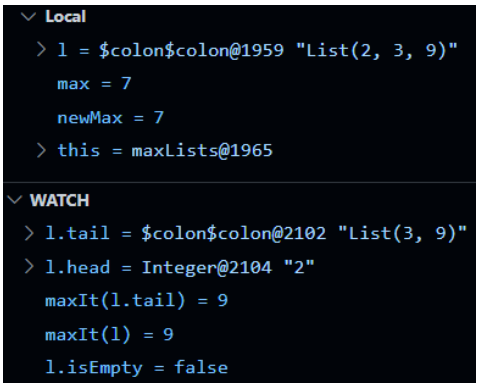


Figura 11: Debug maxIt 4

Ya que el max sigue siendo 7 y el condicional se paso de esa manera, ahora pasa poner el valor de 7 en el newMax que será usado despues.

Figura 12: Debug maxIt 5

Nuevamente reduce la lista una vez más con el llamado recursivo lo que ahora deja a (3, 9) lo que hará que una vez más el condicional se pase para lo siguiente

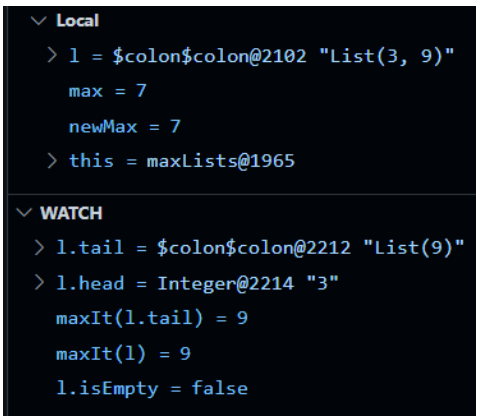


Figura 13: Debug maxIt 6

En esta etapa, la lista se ha reducido a (9) después de procesar el primer elemento. El valor máximo sigue siendo 7. La cola de la lista ahora es vacía (Nil), y la cabeza es 9. Al evaluar maxIt sobre la cola

vacía se obtiene 0, pero maxIt sobre la lista completa aún resulta en 9, que será el máximo final. La lista aún no está vacía en este punto.

Figura 14: Debug maxIt 7

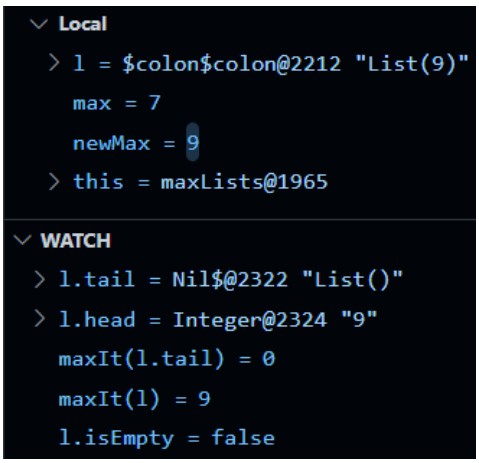
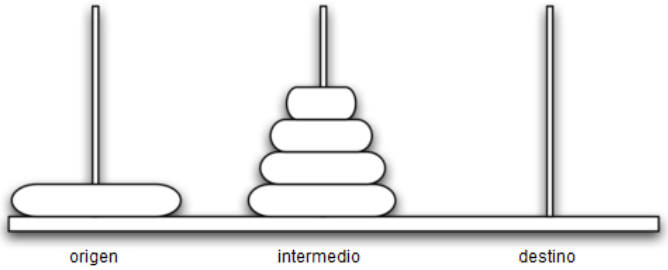


Figura 15: Debug maxIt 8

La función termina con una lista vacía (Nil). El max final es 9, maxIt(l) devuelve 0 para la lista vacía, pero el resultado global del proceso es 9, el máximo encontrado en toda la lista original. Este proceso ilustra cómo maxIt recorre recursivamente la lista, actualizando el valor máximo en cada paso hasta encontrar el número más grande en toda la secuencia.

Figura 16: Debug maxIt 9

1.2. Torres de Hanoi



1.2.1. Función para minimo de movimientos movsTorresHanori()

Ejemplo para movsTorresHanoi(3)
Empieza la función declarando n = 3 que es la cantidad de discos.

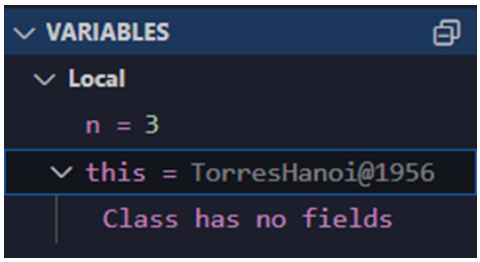


Figura 17: Debug movsTorresHanoi1

Como no se cumple la condición de que n == 1, se haces movTorresHanoi(3-1) y se vuelve a ejecutar la función, esta vez con n = 2.

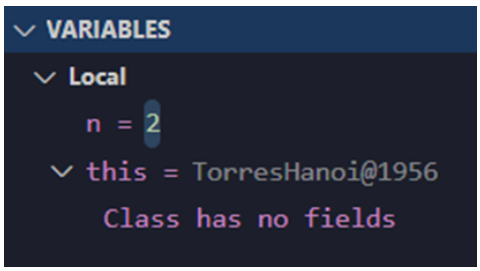


Figura 18: Debug movsTorresHanoi2

Ya que sigue sin cumplirse la condicón se aplica nuevamente movTorresHanoi(2-1) y se vuelve a ejecutar la función, esta vez con n = 1 y se evalua la función de nuevo.

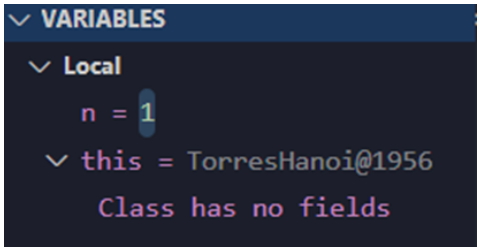


Figura 19: Debug movsTorresHanoi3

Cuando se evalua la función con n = 1 entra en la primera condición y devuelve un BigInt '1'

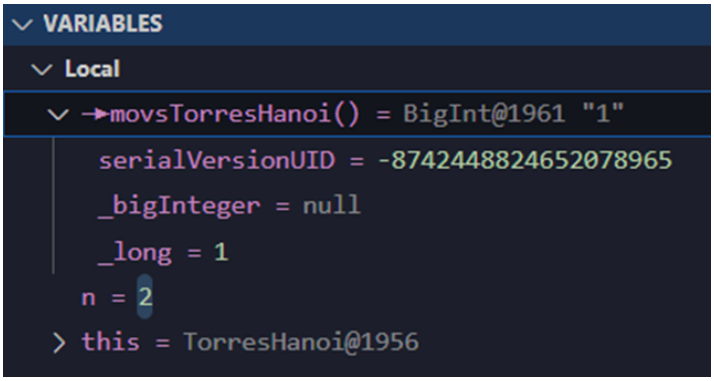


Figura 20: Debug movsTorresHanoi4

La solución a la operación 2 * 1 + 1 = 3 por lo que se determinar que n = 3.

Figura 21: Debug movsTorresHanoi5

Se resuelve la operación 2 * movTorresHanoi(3) + 1 y se obtiene el minimo número de movimientos para solucionar las torres, en este caso '7'.

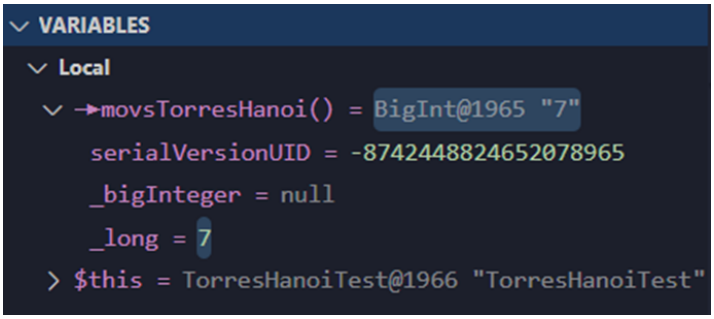


Figura 22: Debug movsTorresHanoi6

1.2.2. Función de la lista de movimientos torresHanoi(n, t1, t2, t3)

Ejemplo para torresHanoi(3, t1, t2, t3)

La función empieza definiendo los valores de las variables, n = 3, t1 = 1, t2 =2, t3= 3

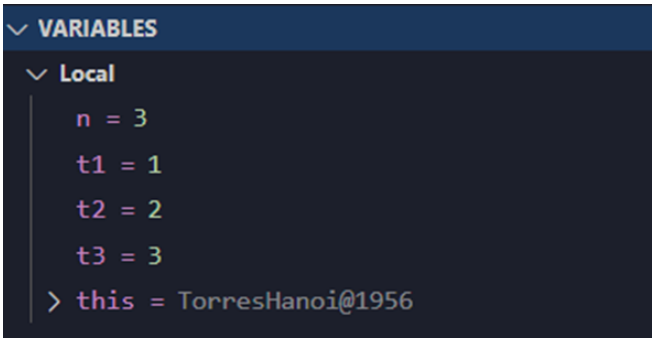


Figura 23: Debug torresHanoi1

Como 3 != 1, se realiza el primer movimiento restándole 1 a n y mueve el disco de t1 a t2 usando a t3 como auxiliar, es decir intercambia el valor de t2 y t3.

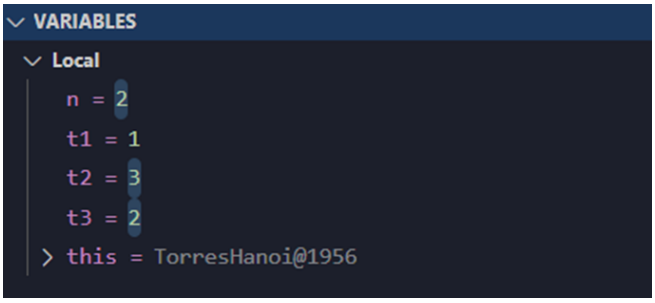


Figura 24: Debug torresHanoi2

Como 2 sigue siendo diferente de 1 se resta nuevamente y al cumplirse 1 == 1 se restablecen las otras variables.

Figura 25: Debug torresHanoi3

La función guarda el movimiento en una lista 'List((1,3))' y se retoma el valor de n = 2 porque aún quedan 2 discos por mover así que se vuelve aplicar movimiento 1

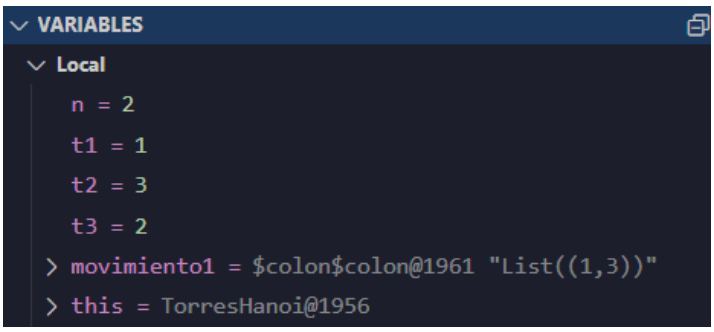


Figura 26: Debug torresHanoi4

Se aplica el movimiento2 y se guarda en la lista el siguiente movimiento 'List((1,2))'

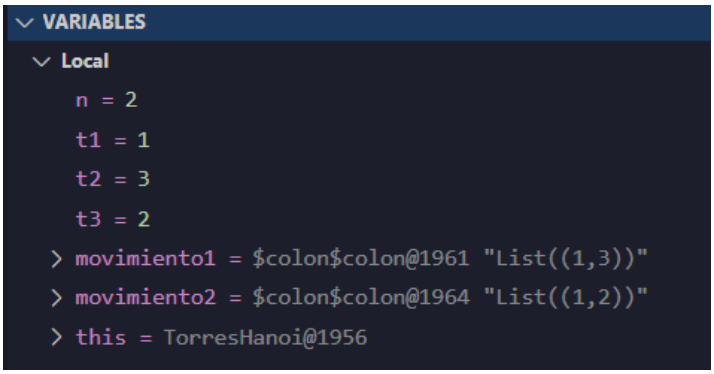


Figura 27: Debug torresHanoi5

Se aplica el movimiento3 restando la ficha que ya se coloco y cambiando las variables t1 = 3, t2 = 1, t=3 = 2 y cómo n = 1 se guarda el movimiento en la lista 'List(((3,2))' se suman los movimientos anteriores y se van guardando en la lista 'List((1,3)(1,2)(3,2))' como se muestra en las figuras 28, 29 y 30.

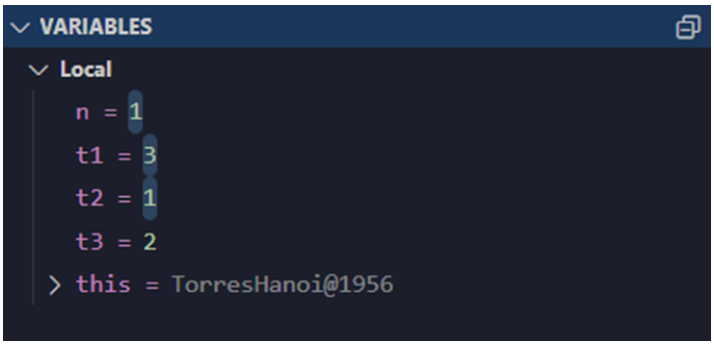


Figura 28: Debug torresHanoi6

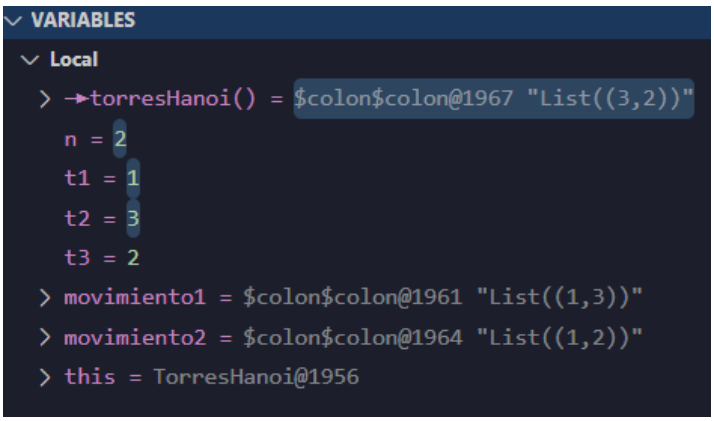


Figura 29: Debug torresHanoi7

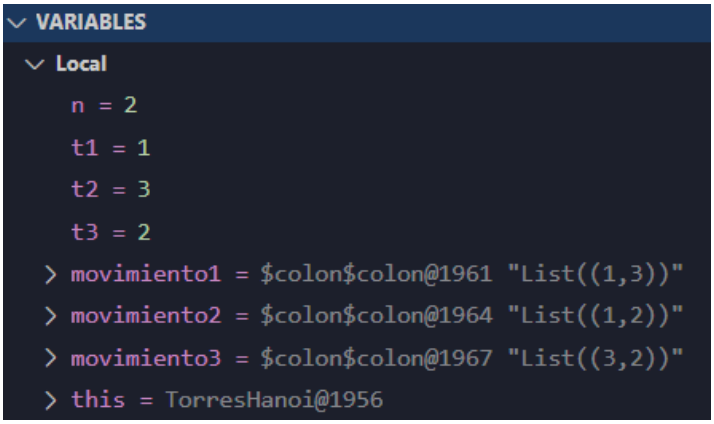


Figura 30: Debug torresHanoi8

La variables vuelven a reiniciarse y se conservan los movimientos anteriores en la lista. Se vuelven aplicar el movimiento 1 y 2, restándole nuevamente a n 1 hasta que sea n = 1. Se guardan los movimientos y así sucesivamente con el resto de pasos.

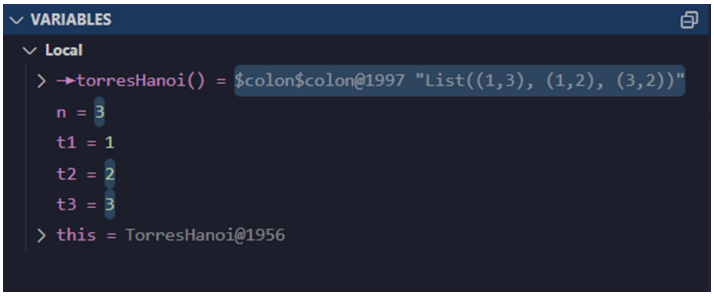


Figura 31: Debug torresHanoi9

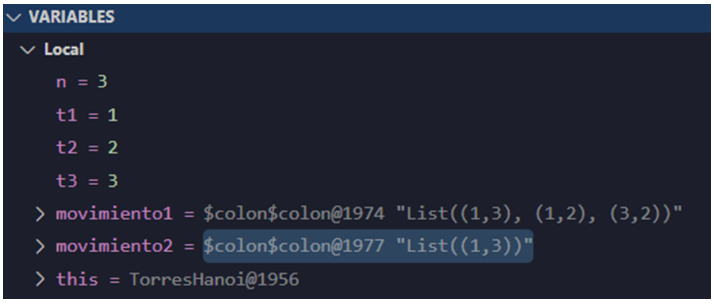


Figura 32: Debug torresHanoi10

Figura 33: Debug torresHanoi11

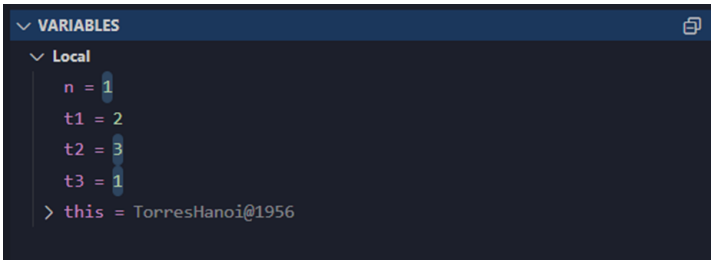


Figura 34: Debug torresHanoi12

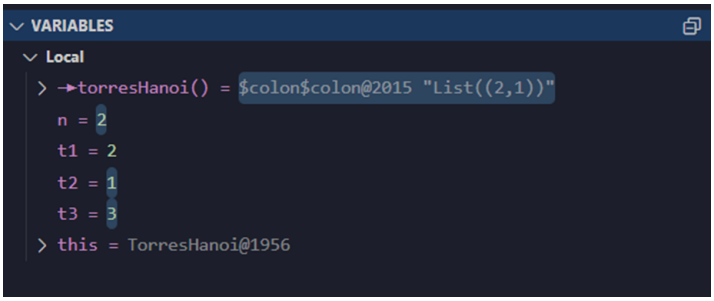


Figura 35: Debug torresHanoi13

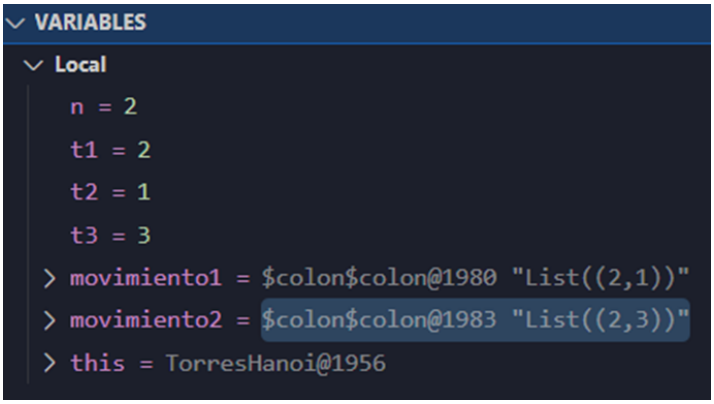


Figura 36: Debug torresHanoi14

Figura 37: Debug torresHanoi15

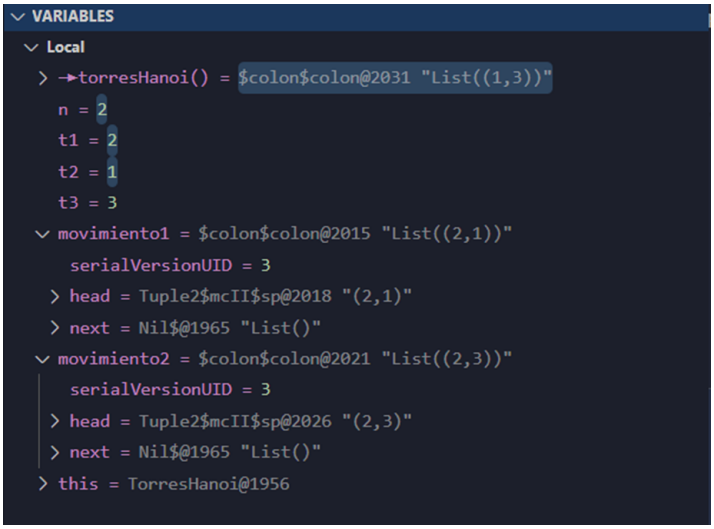


Figura 38: Debug torresHanoi16

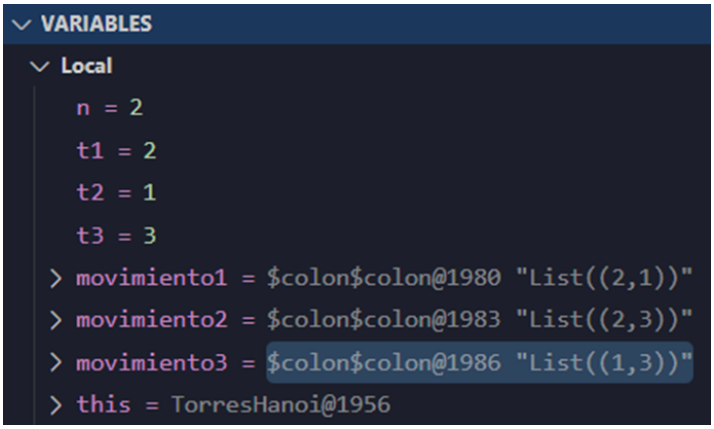


Figura 39: Debug torresHanoi17

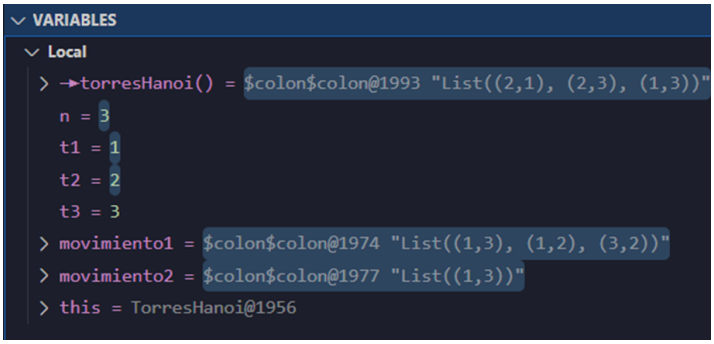


Figura 40: Debug torresHanoi18

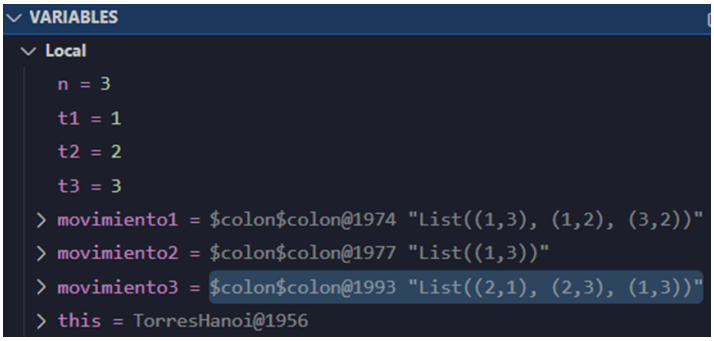


Figura 41: Debug torresHanoi19

Se comparan los pasos generados con el test ya preparado y se confirma que todos los pasos son correctos.

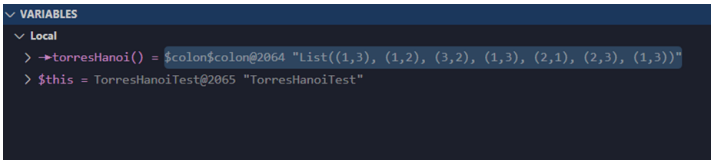


Figura 42: Debug torresHanoi20

2. Informe de Corrección

2.1. Argumentación de Corrección

2.1.1. Función maxLin(List(n))

Se implemento la Función **maxLin** que recibe una lista con n valores y que debe de obtener el mayor valor en la lista recibida. Sea **maxLin** el siguiente programa en Scala:

```
def maxLin (l: List [Int]): Int = {
    if(l.isEmpty) 0
    else if(l.head > maxLin(l.tail)) l.head
    else maxLin(l.tail)
}
```

Vamos a tener que demostrar que $\forall n \in N/0 : maxLin(List(a_1, a_2, ..., a_n) == f(List(a_1, a_2, ..., a_n))$

- **Caso Base:** n = 1

$maxLin(List(a_1)) \rightarrow$ if $List(a_1).isEmpty$ then 0
else if $List(a_1) > maxLin(list())$ then $List(a_1).head \rightarrow a_1$
else $maxLin(list())$

Por lo que f(List(1)) = 1

- **Caso Recursivo:** $n = k + 1, n \geq 1$

Vamos a tener que demostrar que:

$$\forall n \in N/\{0\} : maxLin(List(b_1, b_2, .., b_1) == f(List(b_1, b_2, ..., b_k)) \rightarrow$$
$$maxLin(List(a_1, a_2, ..., a_{k+1}) == f(List(a_1, a_2, .., a_{k+1}))$$

$maxLin(List(a_1)) \rightarrow$ if $List(a_1).isEmpty$ then 0

else if $List(a_1) > maxLin(list())$ then $List(a_1).head \rightarrow a_1$
else $maxLin(list())$
$\rightarrow List(a_1) > maxLin(List(b))$

Hay dos posibilidades:

- Si $List(a_1) > maxLin(List(b)) = a_1$, entonces $a_1 \geq b$ y $a_1 ==$ f(List$(a_1, a_2, ..., a_{k+1})$)
- Si $List(a_1) > maxLin(List(b)) =$ b, entonces $b \geq a_1$ y b == f(List$(a_1, a_2, ..., a_{k+1})$)

Por lo que maxLin(n) == f(n)

2.1.2. Función maxIt(List(n))

Se implemento la Función **maxIt** que recibe una lista con n valores y que debe obtener el mayor valor en la lista pero esta vez por medio de la recursión de cola. Sea **maxIt** el siguiente programa en Scala:

```
def maxIt (l: List [Int] ) : Int = {
    @tailrec
    def maxItAux (l: List [Int] , max: Int ) :Int = {
        if (l.isEmpty) max
        else {
              val newMax = if (l.head > max) l.head
              else max
              maxItAux(l.tail, newMax)
            }
    }
    if(l.isEmpty) 0
    else maxItAux(l, l.head)
}
```

En este caso la función posee diferentes estados que se van a usar para la implementación de su proceso:

- S el cual representa el estado de iteración que es llama a maxItAut y que recibe $S = (l, max)$ en el que l es una lista $(List(a_i, a_{i+1}, ..., a_k))$
- El estado inicial conocido como S_0 tiene los siguientes valores $(S_0(l.head, l.tail) = (a_1, List(a_1, ..., a_k)))$
- El estado final debe de ser debe de ser cuando l quede en una lista vacía tal que así $S = (l, max)$
- La condición que cumple todo estado sería el siguiente: $Inv(l, max) \equiv l = List(a_1, a_2, ..., a_k) \wedge max = f(List(a_1, a_2, ..., a_{i-1}))$
- Transformar $(l, max) = (l.tail, nmax)$ en el que $nmax = max$ si $max \geq l.head$ y $nmax = l.head$ en caso de que no

Demostración

- $Inv(S_f) \rightarrow S_f == f(a)$

$$Inv((l(), max)) \rightarrow max == f(list(a_1, a_2, .., a_k))$$

- $Inv(S_0)$

$$S_0(l(a_2, a_3, ..., a_k), a_1) \rightarrow a_1 == f(list(a_1))$$

- $(Si \neq sf \wedge Inv(si)) \rightarrow Inv(transformar(Si))$

  $!(l.isempty) \wedge l = (List(a_1, a_2, ..., a_k) \wedge max = (f(List(a_1, a_2..., a-1)))\ l.tail = List(a_1, a_2, ..., a_k) \wedge nmax = (List(a_1, ..., a_i))$

2.1.3. Función movsTorresHanoi(n)

Se implemento la Función **movsTorresHanoi** que recibe un número 'n' que es el número de discos y calcula la cantidad mínima de movimientos para llevar los discos de la torre origen al destino. Sea movsTorresHanoi el siguiente programa en Scala:

```
1    def movsTorresHanoi ( n : Int ) : BigInt = {
2        if (n == 1) BigInt(1)
3        else 2 * movsTorresHanoi(n - 1) + 1
4    }
```

Vamos a demostrar que $\forall n \in N : movsTorresHanoi(n) == 2^n - 1$

- **Caso base:** Si $n = 1$

  Sabemos que solo necesita 1 movimiento cuando es una sola ficha, por lo que:

  $$movsTorresHanoi(1) \rightarrow if(1 == 1)\quad BigInt(1)\quad else\quad 2 *\quad movsTorresHanoi(n-1)\quad +\quad 1$$

  Por otro lado, demostrado matemáticamente $f(1) = 2^1 - 1 = 1$

- **Caso recursivo:** Si $n = k + 1, n > 1$ entonces hay que demostrar: $movsTorresHanoi(k) == f(k) \rightarrow movsTorresHanoi(k+1) == f(k+1)$

$$movsTorresHanoi(k+1) \rightarrow if(n+1 == 1)\quad BigInt(1)\quad else\quad 2 *\quad movsTorresHanoi((k+1)-1)+1$$

$$movsTorresHanoi(k+1) \rightarrow if(n+1 == 1)\quad BigInt(1)\quad else\quad 2\quad *\quad movsTorresHanoi(k)+1$$

Para tres discos en la formula matemática seria $2^3 - 1 = 7$ y con la formula que se plantea en el código $2 * 3 - 1 = 7$ Así que concluimos por inducción que $\forall n \in N : movsTorresHanoi(n) == 2^n - 1$

2.1.4. Función torresHanoi (n, t1, t2, t3)

Esta función genera la secuencia de movimientos para mover n discos de la torre t1 a la torre t3 utilizando torre t2 como auxiliar. Sea **torresHanoi** el siguiente programa en Scala:

```
def torresHanoi ( n : Int , t1 : Int , t2 : Int , t3 : Int) : List [(Int , Int)] =
 {
    if (n == 1) {
        List((t1, t3))
    } else {
        // Mover los n-1 discos de t1 a t2 usando t3 como auxiliar
        val movimiento1 = torresHanoi(n - 1, t1, t3, t2)
        // Mover el disco n de t1 a t3
        val movimiento2 = List((t1, t3))
        // Mover los n-1 discos de t2 a t3 usando t1 como auxiliar
        val movimiento3 = torresHanoi(n - 1, t2, t1, t3)
        movimiento1 ++ movimiento2 ++ movimiento3

    }
}
```

Caso base: Si $n = 1$

Para n = 1 la función genera un único movimiento de t1 a t3 ya que no es necesario pasar por t2 para cumplir el objetivo y devuelve una lista con los movimientos realizados.

$$torresHanoi(1, t1, t3)$$
$$if(1 == 1) \rightarrow List((1, 3))$$

Caso recursivo: Si $n = k + 1, n > 1$

Para $n = k + 1$, el proceso recursivo se puede describir en términos de las siguientes tres etapas:

1. **Mover k discos de $t1$ a $t2$** utilizando $t3$ como auxiliar. Esto se describe con la llamada recursiva $torresHanoi(k, t1, t3, t2)$, que por hipótesis inductiva toma $M(k) = 2^k - 1$ movimientos.

2. **Mover el disco más grande** (el disco $k+1$) de $t1$ a $t3$. Este movimiento requiere exactamente un paso adicional.

3. **Mover k discos de $t2$ a $t3$** utilizando $t1$ como auxiliar, lo cual se resuelve con la llamada recursiva $torresHanoi(k, t2, t1, t3)$, que también toma $M(k) = 2^k - 1$ movimientos.

El número total de movimientos para mover $n = k + 1$ discos se expresa como:

$$M(k + 1) = M(k) + 1 + M(k)$$

Sustituyendo $M(k) = 2^k - 1$, obtenemos:

$$M(k + 1) = (2^k - 1) + 1 + (2^k - 1) = 2 \cdot 2^k - 1 = 2^{k+1} - 1$$

Por lo tanto, la recurrencia demuestra que el número mínimo de movimientos para $n = k+1$ discos es $M(k + 1) = 2^{k+1} - 1$.

2.2. Casos de Prueba

2.2.1. Máximo de una lista de enteros

Este ejercicio implementa pruebas para verificar la función **maxIt** de la clase **maxList**, que busca el número mayor en una lista de enteros. Se realizan cinco pruebas: en listas como **List(7, 2, 3, 9)** y **List(812, 7362, 99)**, se verifica que el mayor número es 9 y 7362, respectivamente. También se prueba con una lista vacía, esperando como resultado 0. Todas las pruebas usan la función **assert** para comparar el valor esperado con el obtenido, asegurando el correcto funcionamiento de la función.

2.2.2. Torres de Hanoi

Para este ejercicio se realizan dos funciones que se comprueban por medio de 5 test para cada una. Para la primera función **movsTorresHanoi** que determina la mínima cantidad de movimientos para completar la torre se comprueba cuando se tienen 3, 4, 6, 8 y 64 discos, los resultados se comprueban con la formula ya mencionada. Para la segunda función **torresHanoi** se comprueban las listas de movimientos cuando hay 3, 4, 5, 6 y 8 discos. En este caso se comprueban los movimientos en un emulador de las torres hecho en geogebra: Torres Hanoi

3. Conclusiones

Al realizar este taller de funciones y procesos, no solo tuvimos la oportunidad de practicar la recursividad y la iteración, sino que también comprendimos mejor la importancia de estas técnicas en el desarrollo de algoritmos eficientes. Asimismo, resaltamos el valor de las pruebas exhaustivas y el razonamiento matemático como herramientas esenciales para garantizar un código robusto y confiable.

- **Importancia de la Recursión:** La recursión es una técnica clave en programación, ya que permite abordar problemas complejos de manera más organizada y sistemática. Al descomponer un problema en subproblemas más simples, la recursión ofrece una solución estructurada y, en muchos casos, más intuitiva, facilitando el proceso de razonamiento y solución.

- **Pruebas y Verificación:** La implementación de algoritmos debe ir acompañada de pruebas rigurosas para validar su correcto funcionamiento. Las pruebas exhaustivas con diferentes casos de uso garantizan que el algoritmo sea preciso y eficiente, asegurando que los resultados sean consistentes y predecibles bajo diversas condiciones.

  Un desafío particular surgió al desarrollar las pruebas para las Torres de Hanoi, ya que el número de movimientos necesarios aumenta exponencialmente con el número de discos. Comparar los movimientos generados por el programa con los esperados resultó complejo, debido a la cantidad significativa de pasos involucrados, especialmente para 6 y 8 discos. Sin embargo, la verificación exhaustiva a través de pruebas permitió asegurar que el algoritmo funcionaba correctamente, garantizando que los movimientos realizados eran los óptimos.

- **Eficiencia de los Algoritmos:** El análisis de la eficiencia de un algoritmo, tanto en términos de tiempo de ejecución como en el uso de recursos, es fundamental. En este sentido, evaluar cuándo utilizar recursión o iteración es esencial para optimizar el rendimiento de las soluciones. La recursión, en particular, permite resolver problemas que de otra forma serían más difíciles de abordar, mientras que la iteración puede ofrecer ventajas en ciertos casos donde el control explícito del flujo es más eficiente.